University of Illinois at
Urbana-Champaign

ECE 445 Team #14

Child Development Sensor
Individual Progress Report

Yang An Tang (ytang46)

November 5, 2017

Contents

1 Introduction **1**
- 1.1 Individual Objectives . . . 1
- 1.2 Overal Progress . . . 1
- 1.3 Individual Progress . . . 2

2 Design **2**
- 2.1 Bluetooth Low Energy . . . 2
- 2.2 Sensor Interface . . . 5
- 2.3 WiFi . . . 7

3 Conclusion **9**
- 3.1 Timeline . . . 9
- 3.2 Ethics and Safety . . . 9
- 3.3 Self-assessment . . . 9

1 Introduction

1.1 Individual Objectives

The overall goal for our project is to develop a system that enables researchers to study a child's development through accurate sensor data collection of heart rate and audio recording. This system will be a wireless device that helps the researchers overcome human-error limitations during experiments. The system comprises of a sensor node with most of the low-powered modules will communicate with a main hub that will house the microprocessor for heavy computations, for an added bonus multiple sensor nodes can connect to a single main hub for easy scaling of experiments carried out. To breakdown this project, we have decided to split the development work into three main portions. That is sensor development, communications (which include Bluetooth ,WiFi, and sensor interface), and power electronics. I have been tasked to handle the communications part of this project, sensor development (Eu Hong Woo) and power electronics (Xiang Wen) will be led by my other teammates.

To further breakdown my section of the work into submodules, there are three main tasks to carry out. They are the implementation of Bluetooth Low Energy (BLE), WiFi protocols, as well as the general interface between sensor node and main hub. All of these submodules will need to be robust enough to work all of the time and interface well with the other components of the project, so heavy testing and verifications have to be done on my part.

OPTIONAL: Develop a simple mobile application or web application that pulls data down from the server, eliminating the need for researchers to be present during experiments. Also, allowing for data collection at home in a more natural environment as long as WiFi is available.

1.2 Overal Progress

Currently the overall team progress is on track based on our proposed project timeline except for the PCB layout which we originally planned for it to be done by the 9th of November. Our alternative for now is to lay out a simple shield for the sensor node that would house all off-the-shelf modules which include the buck converter, ECG sensor, MEMS microphone, and the Bluetooth module. For a second revision of the board, we will be striping down all of these modules such that it has only the ICs and the necessary components to drive it on the board.

This way we will be able to minimize the PCB footprint. The reason for creating an initial PCB shield is to provide a working system just in case we are not able to debug and get the final revision working on time. We are prioritizing on getting the proposed functionalities fully implemented and ready to go.

1.3 Individual Progress

Up till this point, **Bluetooth Low Energy (BLE) is fully functioning and tested**. More on the function and validation details in 2.1. The next task is to develop an ADC to interface with the ECG sensor's analog output. For simplicity, the Bluetooth module's on-board ADC will be used for development purposes to make sure reading from ECG sensor works. Once that has been established and if time permits, a standalone ADC will be built onto the sensor node PCB interfacing the ECG sensor analog output to the Bluetooth module's digital I2C input.

The last submodule to be developed is the WiFi implementation. We chose a Bluetooth and WiFi combo module for both the sensor node and main hub, WiFi is disabled through software for the sensor node to reduce power draw. Most of WiFi development will be writing firmware code that reads data periodically from the microcontroller on the main hub then sending TCP packets using MQTT protocols to an AWS server. For safety, data will be copied into some form of local storage (e.g. USB stick, or on-board memory) in the event that WiFi does not function as expected.

2 Design

2.1 Bluetooth Low Energy

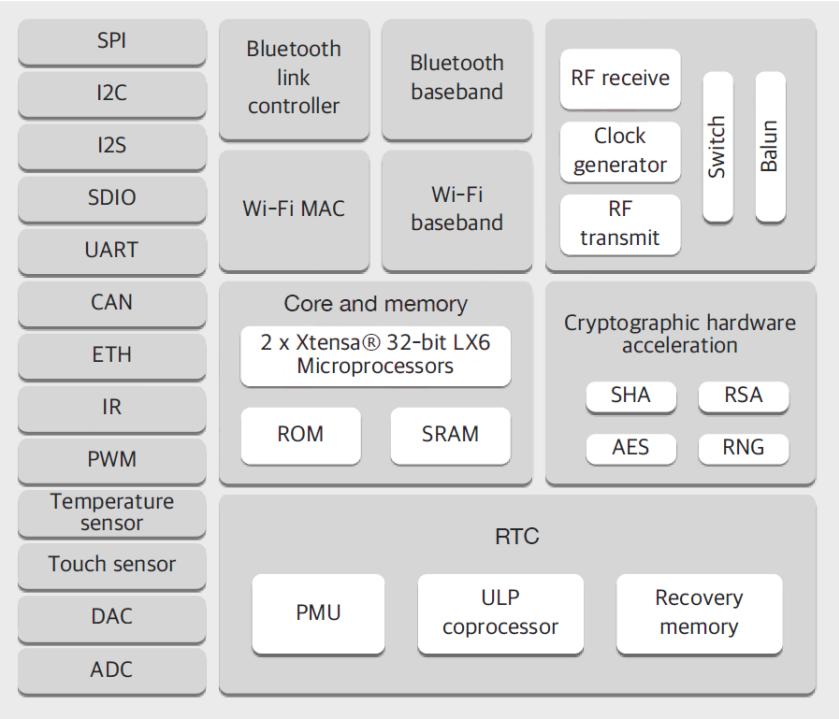

Figure 1: ESP32 Block Diagram

Since designing a Bluetooth capable IC from scratch is not within scope of this course and also not within my capabilities I have resorted to use an off-the-shelves module which is the ESP32

Bluetooth and WiFi module from Espressif Systems. Initially, we chose to use different Bluetooth modules for the sensor node and main hub respectively because on the sensor node we would not need a WiFi capable module. But I have since resorted to using the same ESP32 board for both, having similar hardware saves on hardware and software development time. The block diagram above in *Figure 1* shows what the ESP32 hardware has to offer.

Firstly, to make sure that the ESP32 hardware provides **sufficient throughput/bandwidth** for our purposes, a simple calculation was done. Each `Write Command` allows 20 Bytes of data according to specifications. These packets do not provide any application level acknowledgement of sending, and hence can be queued. Next, the lowest interval allowed is 7.5 ms and the maximum number of packets per connection event is 6.[5]

Given number of packets per event as n, connection interval T, and a single command allows B number of Bytes, the throughput is

$$Throughput = n \times B \times \frac{1}{T} \tag{1}$$

where $n = 6$, $T = 7.5\ ms$, and $B = 20$, total throughput is

$$Total\ Throughput = 6 \times 20 \times \frac{1}{0.0075\ s} \tag{2}$$

$$= 16\ kB/s = 128\ kbps \tag{3}$$

which is more than enough for our system's usage. Since only the microphone data will be using up most of the bandwidth as it is running at 8 kHz to be able to capture audible speech.

Range and sensitivity is also another major consideration for our purposes. Since BLE on sensor node will be transmitting and BLE on main hub will be transmitting and receiving respectively, it is important that the line-of-sight range is at least the span of an average size room in order for the connection to work. According to the ESP32 datasheet, it has a line-of-sight range of 150 meters which is sufficient.

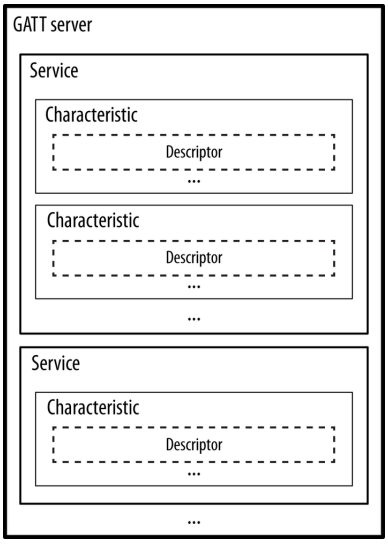

Figure 2: BLE GATT Server Structure

For the ESP32 firmware, a GATT server had to be setup according to BLE protocols for the sensor node and main hub to communicate. GATT server specifications can be found at [6].

The main hub in this case will be referred to as the master and the sensor node as the slave. In order for the slave to write to the master, a specific Service UUID which acts as the "address" at which both the master and slave will be communicating at. The master will have to setup a GATT Server that hosts the specific Service UUID so that the slave can connect to it. Within a given service, there can be more than one characteristic which in this case denotes data from sensors. Since we have both ECG and microphone sensors, there will be two characteristics with unique given Characteristic UUIDs under the same Service. *I have successfully implemented the slave and master handshaking described above using the correct BLE protocols.*

During development there were a few bugs namely scanning takes a long time due to too many Bluetooth devices in the area and the written embedded application keeps dumping. To fix the first problem, hardware scan rate was increased through the BIOS. For the second problem, the user-program stack size was increased and heavy standard C libraries were omitted.

Below is the requirements and verifications table.

Requirement	**Verification**
Experimental throughput of 96 *kbps* which consist of ECG sensor output (32 *kbps*) and audio recording ($\frac{32\ bits}{0.5\ ms} = 64\ kbps$).	Wrote a test case that sends dummy data using the specified frequencies and checks to see if any packets of data were dropped.
Master and slave completes handshaking protocol within 15 ms to prevent stack overflow on slave since each server discovery eats stack space.	Exception handler written to exit program when 15 ms is over otherwise print connection established to serial stream and continue running.

```
/* BLE Server initialization, advertising server, acknowledging client connection */
BLE.init()
BLE.create(server)

/* create server with service of "serviceUUID" and characteristic of "charUUID" */
BLE.server.create(serviceUUID)
BLE.server.create(charUUID, value)

/* start advertising server */
BLE.advertising(start)
```

Figure 3: Pseudocode for BLE Master.

```
/* BLE Client initialization, connecting to server, and writing to specific service */
BLE.init()
while(1)
    if(BLE.scan(serviceUUID))
        BLE.create(client)
        /* check to make sure service exists */
        if(!BLE.connect(serviceUUID->address))
            print "Failed to connect to server"
            return
        /* writes to service->characteristic->value */
        BLE.write(address->charUUID, value)
    else
        print "Failed to find server"
```

Figure 4: Pseudocode for BLE Slave.

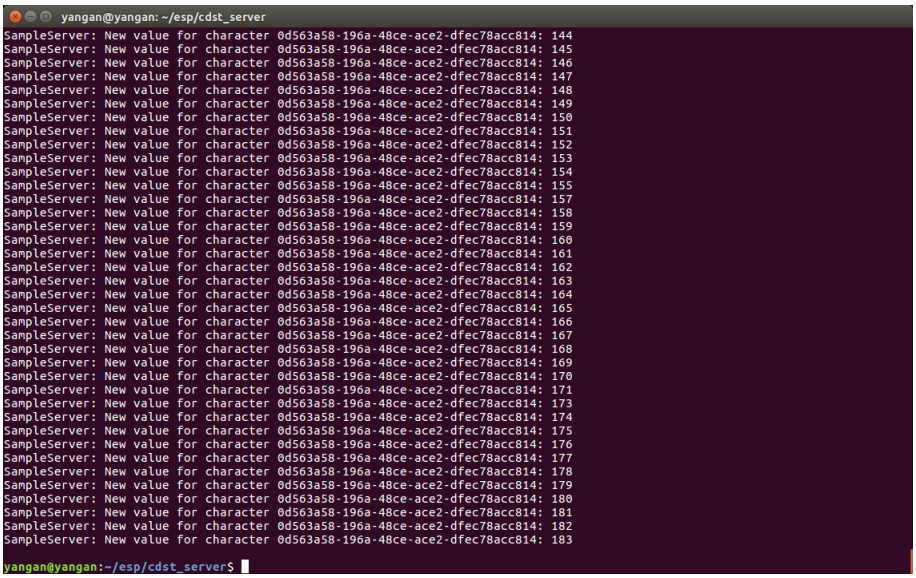

Figure 5: BLE Master successfully hosting server and receiving each packet from Slave.

In *Figure* 3 is the pseudocode for initializing and advertising a BLE GATT server. Once initialized, the program waits for an incoming connection request from the specified service UUID. On the other hand in *Figure* 4, is the pseudocode for initializing a slave then scanning for the specified service UUID before requesting to connect.

2.2 Sensor Interface

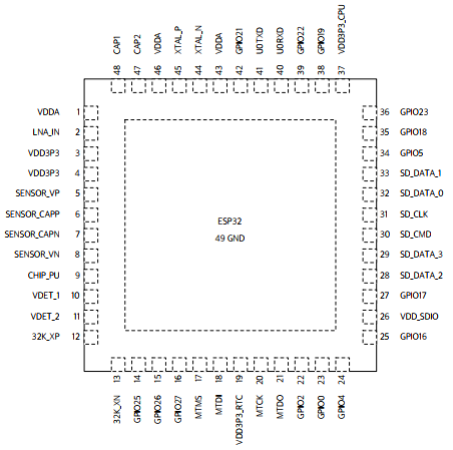

Figure 6: ESP32 Pinout Diagram [5]

The pins of interest in *Figure* 6 are GPIO32-39 which are used by the on-board ADC, GPIO 25-26 used by I2S input, and for I2C any two GPIO pins can be programmed to act as SDA and SCL pins respectively.

Firstly, the **ADC** which connects the ECG sensor which gives an analog output to the ESP32 should ideally be built rather than implemented by software on the ESP32 chip for added complexity. But for faster development, the on-board ESP32 ADC will be used to prototype before moving on to designing an ADC. To use the on-board ADC, there are 8 pins (GPIO32-39) that

are used since it is an 8-channel ADC but only one will be wired in our case.

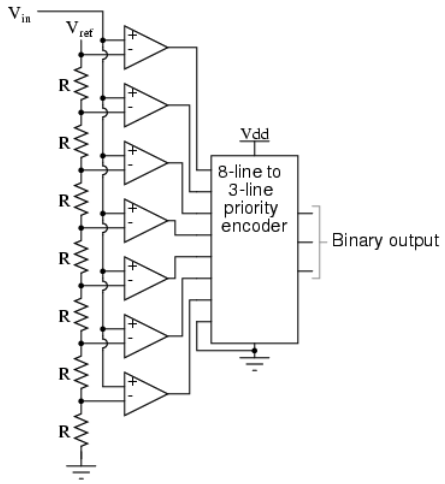

Figure 7: 3-bit Flash A/D Converter [7]

To design an ADC, I have settled on a parallel A/D converter as it is the simplest to implement. It is simply a series of comparators, each comparing input signal to a chosen reference voltage. Then, the comparator output connects to the inputs of a priority encoder which produces a binary output. Above in *Figure* is a 3-bit flash A/D converter example. A flash A/D converter is also the most efficient in terms of speed, the limiting factor being only the gate propagation delays of the encoder. But, unfortunately it requires the most number of components to build. The number of comparator op-amps doubles with each additional output bits,

$$3\ bits = 2^3 - 1 = 7\ comparators \tag{4}$$

$$4\ bits = 2^4 - 1 = 15\ comparators \tag{5}$$

$$5\ bits = 2^5 - 1 = 31\ comparators \tag{6}$$

For our use an 8-bit ADC would need 255 comparators which is not ideal to design. **ToDo:** *If time permits, research other ADC design and implementations.*

Since the on-board ADC only outputs at 12-bits, each integer output value will have to be bit-aligned to 32-bits. This will reduce the resolution of the output waveform when plotting according to the needs of the sponsoring researchers. **ToDo:** *Will need to test accuracy and validity of on-board ADC output.*

Below is the requirements and verifications table assuming that the on-board ESP32 ADC is used.

Requirement	**Verification**
Able to output 32-bit integer values for each analog read since ECG waveforms can be plotted given a sample rate of at least 500 Hz.	**ToDo:** Write a test script that samples the voltage readings from the ECG sensor and plots out the waveforms. Check the waveform against a reference waveform.

Next is the **I2S input** from the MEMS microphone which is being done by Eu Hong Woo. He is currently writing the firmware initialization for I2S on the ESP32 board. The board has two I2S peripherals [5] but only one will be configured via the I2S driver to support reading input from the microphone. He will also be heavily testing the I2S interface to ensure full functionality.

2.3 WiFi

WiFi will only be enabled on the ESP32 chip that is on the main hub and connected to the microcontroller. The general idea is for the ESP32 chip to read data over serial using I2C protocols from the microcontroller, then sending the TCP packets using MQTT protocols to the AWS IoT server. To establish MQTT connection, the WiFi client must first request connection with a unique client ID, once connected the client can start publishing to the specific topic that is setup by the server.

ToDo: *Will have to first setup AWS IoT server with MQTT protocols enabled and initialized. More specifically, a unique topic for each data points (ECG and audio) will be advertised. This can be done through the AWS online console or through* `ssh`*. Then, the firmware for the ESP32 on the main hub will have to be modified to include WiFi specific function declarations and implementations together with the AWS and MQTT API calls.*

```
/* Pseudocode: WiFi connect to AP and send to AWS IoT using MQTT protocol */

/* Initialize WiFi event handler and connect to AP */
wifi.init()
if(!wifi.connect(AP))
    print "Failed to connect to access point: %x", AP
    return

/* Connect and initialize MQTT channel */
if(!AWS.connect(clientID))
    print "Failed to connect to AWS server of ID: %x", clientID
    return
AWS.init()

/* Read from microcontroller using I2C protocol
 * Publish or write to specific MQTT topic
 */
while(1)
    value = I2C.read(microcontroller)
    AWS.publish(value)

AWS.disconnect()
```

Figure 8: Pseudocode for WiFi algorithm that publishes to AWS Server.

The above pseudo-algorithm in *Figure* 8 is very simple compared to the actual implementation because it omits most of the configuration declaration such as callback functions, hardware pin settings, drivers, Rx/Tx buffer, NVS structure etc. Callback functions in this case will be called anytime a Tx packet is sent, e.g. to read next value in I2C serial buffer. Since data will be coming in from GPIO pins, these pins have to be setup to accommodate I2C protocols (SDA, SCL pins). Next, the Rx/Tx buffer acts as a queue since the program has to wait for an receive acknowledge before sending the next packet, a buffer queue is needed to maintain First-In-First-Out (FIFO) arrangement.

A problem that I have been facing is when enabling both WiFi and Bluetooth libraries on the ESP32 hardware, the entire embedded firmware seems to crash as if there is not enough stack space for so man libraries. The kernel code runs just fine when only WiFi or Bluetooth is enabled one at a time. Increasing the stack space works only on some occasions and not all the time because every new call to WiFi or Bluetooth class eats up stack space. There are a couple of probable solutions queued up such as going through the libraries one-by-one and omitting files that are not used, creating a intermediate virtual memory map so that every new task will not override kernel code. As a contingency plan if I am still not able to sort this out, I will then just use two separate ESP32 chips, one for Bluetooth and one for WiFi.

Below is the requirements and verifications table.

Requirement	Verification
ESP32 able to properly read from microcontroller GPIO pins using I2C protocol at a rate of 115200 baud or more. This is so that there are enough samples per second to reproduce clear audio recordings.	**ToDo:** Manually send the data length each interval so that ESP32 program can check integrity of received data.
ESP32 able to connect, subscribe, and publish to specified AWS MQTT topic.	**ToDo:** Write AWS Lambda verification script that throws an exception when there is a failed connect or a sudden disconnect for logging and debugging purposes.
AWS MQTT server setup such that it is able to create new topics, delete old topics, receive topic updates, and publish to existing topics.	**ToDo:** Use open-source MQTT client applications to test AWS server by trying to subscribe to existing topic, publish to existing topic, unsubscribe from existing topic.

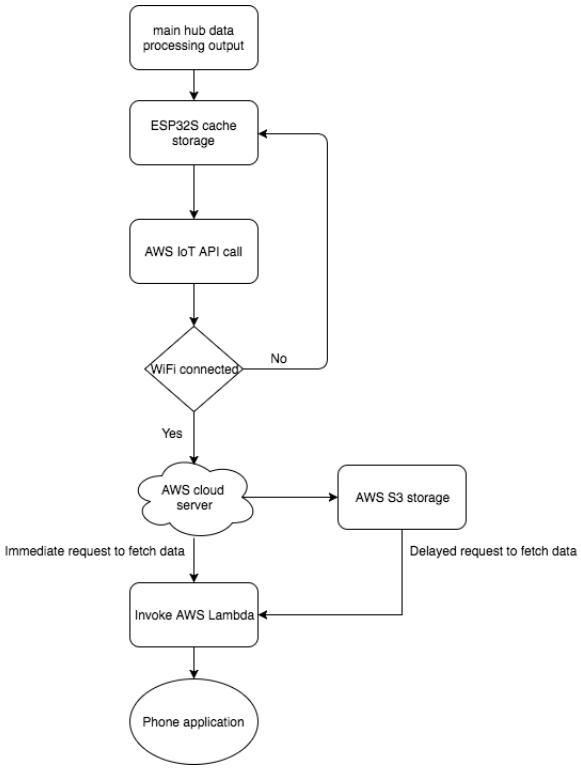

Figure 9: Flowchart for AWS data back-end.

The above flowchart shows the general data pipeline from the main hub microcontroller all the way to the users either through local storage or online server.

OPTIONAL: If I am ahead of schedule, I will work on creating a mobile phone or web application that pulls selected chunks of data from the AWS server. A simple Ionic framework based mobile phone application can be written with back-end API calls to AWS. Ionic is used because instead of developing Android or iOS independently, Ionic allows development in HTML/Javascript which can then be ported over to either mobile OSes.

3 Conclusion

3.1 Timeline

Deliverables	Due Dates
ESP32 firmware to send test data (slave)	Done
ESP32 firmware to host GATT server (master)	Done
ESP32 master and slave communication	Done
PCB rev 1 design and layout	9th November
ESP32 ADC implementation	9th November
ESP32 interface with sensors	16th November
PCB final rev design and layout	18th November
ESP32 working WiFi drivers	25th November
Debug and test PCB	2nd December
Cloud integration with AWS API	2nd December
OPTIONAL: phone application	2nd December

As of today, Bluetooth Low Energy drivers for our program have been fully implemented and tested. Individual checkpoints are to implement ADC, first with ESP32 on-board 12-bit 8-channel ADC or building an off-board ADC. For the sake of time, I have chosen to develop with the on-board ADC first. Next on my schedule is to write working WiFi firmware for the ESP32 main hub program that will serve as the interface to the AWS server. The last thing on my end is the setup and configuration of an AWS IoT MQTT server with unique topics for each data point. Finally, PCB layout, design, and debugging is a group effort.

3.2 Ethics and Safety

ACM Code of ethics stated that we ought to respect the privacy of other [2]. The microphone in the design enables constant recording for research purposes, to protect the privacy of an individual, extra precautions will be taken to ensure the accurate data obtained is protected from unauthorized access or accidental disclosure to inappropriate individuals.[2] This can be achieved by having all the collected and streamed data hardware and software encrypted, allowing only authorized users to decrypt said data. The hardware components mentioned in the Design section comes with built-in hardware encryption accelerators and software encryption that can be realized by first encrypting each data packet before transmitting. Encryption will be enabled concurrently on the server side. All the wireless communication components also comply with the FCC requirements.

To protect the rights and welfare of human subjects in research, we ought to abide to the Institutional Review Board (IRB) which takes oversight of research involving human subjects, and to comply with the Federal Policy for the Protection of Human Subjects and applicable federal laws and regulations [3], we intend to test our system on toddlers towards the end for validation purposes with Professor Nancy McElwain from the College of HDFS. The IRB ensures that appropriate safeguards exist to protect the rights and welfare of research subjects. [4] IRB review process will be led by the reserach coordinators, therefore, we will have all the required approval to carry out testing with proper consent and in accordance to IRB protocol.

3.3 Self-assessment

Based on the above discussions, I feel that I am a little behind schedule. Most of my time is spent on debugging code due to the complexity of verifying low-level hardware code. Once the WiFi program code is done, I will have more time to work on circuit schematics PCB layout. Also,

to increase productivity, I will be dedicating more hours a week so that our team can deliver a working final prototype by the end of the course to the sponsoring researchers.

I am dedicating on average 28 hours a week on this project to keep up with the proposed schedule of weekly tasks. I believe that the workload is very evenly spread out amongst each individual contributor. The blame is on me that I am slightly behind schedule and I will have to make up for it in the coming weeks as testing of the full system prototype in real-world environment will have to be done before the final demo. I do want this project to be a success and for it to continue past the scope of this course.

References

[1] IEEE Code of Ethics [Online]. Available:
http://www.ieee.org/about/corporate/governance/p7-8.html

[2] ACM Code of Ethics [Online]. Available:
https://www.acm.org/about-acm/acm-code-of-ethics-and-professional-conduct

[3] Urbana-Champaign Policy Governing the Use of Human Subjects in Research [Online]. Available:
http://cam.illinois.edu/xi/xi-1.htm

[4] Review Processes and Checklists [Online]. Available:
https://oprs.research.illinois.edu/review-processes-checklists

[5] Espressif Systems (October 2017). ESP32 Datasheet v1.9. Available: http://espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[6] GATT Specifications [Online]. Available:
https://www.bluetooth.com/specifications/gatt/

[7] Flash ADC: Chapter 13 - Digital-Analog Conversion [Online]. Available:
https://www.allaboutcircuits.com/textbook/digital/chpt-13/flash-adc/